COMP 3400 Programming Project : The Web Spider

Due Date: Tuesday, 25 April 2017 (see page 4 for phases and intermediate deadlines)
Worth: 65 points

Introduction

Web spiders (a.k.a. crawlers, robots, bots, search engines) are programs that automatically search the World Wide Web for documents of interest then index those documents based on the spider's purpose. In this assignment you will write a simple spider in Java to profile a given web site.

There are many web spiders available with published source code, and even guides for building web spiders. You are free to study and learn from these, but you are *not* free to use the source code they provide. They are way more comprehensive and complicated than I require for this project and besides it is not that difficult to write yourself because the Java API provides a number of useful classes.

The objective of this assignment is to convince you of the relative ease with which useful HTTP client applications can be designed and implemented.

The main procedure must be contained in class `Spider`. Name other classes as you choose.

You may work either individually or with one partner.

The Challenge.

Implement an HTTP client application to explore and profile a web site. For full credit, the application should have a GUI user interface.

User Interface.

The GUI should include the following components:
1. a text field where the user can type in the "root" web page for initiating the search.
2. a text field to display the name of the web page currently being visited. This provides a simple trace and progress indicator.
3. a text field where the user can type the maximum number of web pages to visit. Default is 100.
4. a scrollable text area (use JScrollPane wrapped around a JTextArea) for displaying results at the conclusion of the search.
5. a search button for the user to click, to initiate the search.
6. a clear button for the user to click, to set the text fields and areas to their default values (100, for maximum pages, blank for the others).
7. an exit button for the user to click, to terminate the application.

Search Protocol.

- Your program will use socket connections to communicate with web servers. There are a number of Java API classes that abstract the details away. However you may use sockets directly if you wish.

- The search begins with the web site provided by the user, and is performed recursively on its links (restricted to other pages on the same server).

- For each page requested, you will first issue a HEAD command. If the header information indicates that the file is OK (status code is 200) and it is an HTML file (content type is text/html) and it is on the web site's server, then issue a GET command for the same file. Read the file contents and parse it to find links and references to images. Details below.

- Collect statistics based on file contents. These will be displayed at the conclusion of the search. Links found during the parsing process are similarly explored if not already encountered.

- The search concludes when all relevant links have been explored or the page maximum specified by the user is reached, whichever happens first.

Search Results.

When the search is complete, display a statistical summary (in the text area described above) to include:

1. Number of HEAD requests issued
2. Counter for each status code returned from a HEAD request (e.g. how many 200s, how many 404s, etc)
3. Number of GET requests issued
4. Number of web pages successfully fetched (should match number of GET requests)
5. For pages successfully fetched using GET, report the minimum, maximum, and average:
    a. page length in bytes
    b. number of images per page (count "IMG" occurrences)
    c. number of hyperlinks per page (count "HREF" occurrences)
    d. number of internal links per page (links to pages on same server)
    e. number of external links per page (links to pages on different server)

For #5, don't be concerned with duplicate links. For example, if a page contains three different links to a web page, count all of them. That does **not** mean you should recursively explore all three – this is explained below.

Upon each successful GET, display the page's URL for the trace (GUI component #2 above). This will provide the user some visual feedback during the search.

The Basic Idea.

Your program will get a URL from the text field, contact its HTTP server through a socket, and request the page using the HTTP HEAD command. If this is not successful, which you can tell from the status code, that's it. If it is successful, fetch the page using GET, record the page length in bytes, count the number of image references and hyperlinks, then repeat the process recursively for each hyperlink. Notice that every page request is first done using HTTP HEAD command. Response will be a header. If header status code is 200 *and* content type is text/html *and* the page is on the same server (identified in command line), then issue GET to fetch the page and repeat the process. Otherwise, do not issue the GET. This means your spider will *not* follow links that go to other web servers.

It is imperative that you maintain a data structure of all pages fetched and explored using GET, and imperative that you not GET the same page twice. Doing so will get your program into a cycle of links (there are many of these) and that is neither productive nor terminal.

Details.

1. Use the Swing API For the GUI: `JFrame`, `JTextField`, `JTextArea`, `JButton`, `JScrollPane`, etc. If you haven't done this before, I have a "Need to Know" handout.

2. There are two basic approaches to analyzing HTML content. Both involve using the `URL` and/or `HttpURLConnection` classes to first read the web page into a String. You will find examples of this on the web. After the HTML file has been read into a String, either

   a. use String methods to locate elements and their contents.

   b. parse it into a DOM-like structure for analysis. Classes such as `HTMLEditorKit` and `HTMLDocument` will be useful here. The Jsoup open source Java HTML parser (see jsoup.org) might be an easier alternative.

3. When crawling, beware of cycles! For instance, if you fetch `index.html` and find it has a link to `info.html`, then you fetch `info.html` and it has a link back to `index.html`, then a hyperlink cycle is formed. If you blindly explore every link, you'll get into a never-ending cycle! To avoid it, you have to keep a list of each page that has been already been explored. To do this, add the URL to the list after it has been fetched using GET. Then before issuing any HEAD command, check the requested URL against this list. If already there, do not issue the HEAD. Consider using a hash table. The `URL` class in `java.net` should be useful, and provides a `hashCode` method.

4. Your source code should be adequately documented (self-documenting identifier names, comments, indentation, etc). Be sure to put your name(s) at the top of every file!

5. Other details come to light in the FAQ list below. You must read and understand them.

Schedule and Point Allocations

This project is worth 65 points, about 8% of your grade. Points are allocated both through phased project deadlines and by quality criteria. Point values for each phase are reduced by 10% for each day late (midnight deadline), down to half credit minimum.

Phase 1 – The User Interface
Deadline: Tuesday April 4
Value: 10 points (6 correctness, 1 design, 3 documentation)
Details: GUI Prototype. Contains all the elements of the user interface, appropriately arranged. Buttons may be non-functional, text areas may contain fixed sample text.

Phase 2 – Analyze Initial Web page
Deadline: Thursday April 13
Value: 30 points (24 correctness, 4 design, 2 documentation)
Details: The above user interface, plus the ability to type in a URL and, upon click of the button, to read in the corresponding web page. Upon reading the page, determine its length, number of hyperlinks (without distinguishing external from internal), and number of images, then display those values.

Phase 3 – Crawl
Deadline: Tuesday April 25
Value: 25 points (20 correctness, 3 design, 2 documentation)
Details: Assumes that all the above are completed and functional. Implement remainder of the project requirements. This includes, but is not limited to, recursively following internal links up to the specified limit, calculating the required statistical values, and displaying the final statistical values. See the project assignment for more details.

Explanation of the quality criteria:
Correctness means it functions according to the project specifications.
Design means the code structure follows object-oriented design principles.
Documentation means code comments including your name, self-documenting identifiers, judicious use of white space including indentation (the latter is trivial if you use jGRASP CSD feature).

FAQs

Q: What distinguishes "internal" versus "external" pages?

A: If the URL is a relative pathname, the page is internal. If URL is absolute, extract the host name. The Java `InetAddress` class provides a static method called `getByName`, which takes a `String` containing presumably a host name, and returns its IP address. It returns an object of type `InetAddress`. My suggestion is that you use this at the start to get the IP address of the server specified in the command line. Then for each absolute URL, similarly extract the host name and gets its IP address. If it matches the initial one, the page is internal else it is external. If this causes serious performance problems, you might want to just keep the original server name in a `String` then for each absolute URL extract the host name and compare the strings for equality. This will catch everything except aliases, and should run faster.

Q: Regarding the summary information, it is not clear to me what counts as an image or a hyperlink for statistical purposes.

A: Interpret "number of images on a page" as number of IMG elements.
Interpret "number of hyperlinks per page" as number of HREF attributes of A elements.
Interpret "number of insider hyperlinks per page" as number of internal (see previous question) HREFs that link to HTML files.
Interpret "number of outsider hyperlinks per page" as number of external (see previous question) HREFs that link to HTML files.

Q: How do I know whether an HREF refers to an HTML file or not?

A: You can request information about a page without fetching the page itself, by using the HTTP command HEAD instead of GET. HEAD will return only the header information. Look for the "Content-Type" header line. If the content type is "text/html" then it is HTML, otherwise not. *This means you will issue HEAD commands to external servers, but you will never issue a GET command to an external server.*

Q: Does this mean I'll use HEAD as a filter of sorts?

A: Exactly. Rather than trying to determine the file type based on filename extension, you will request information about that file using HEAD. The response message will indicate the file content type. Besides, filename extensions such as `.asp` or `.php` refer to server side software that generates the HTML page dynamically upon request. Once you get information on an internal file using HEAD, you will issue GET for the same file only if the status code was 200 (Ok) and the content type was text/html.

Q: What if response to HEAD is code 404 (not found)?

A: If you request a file that does not exist, the content type will come back as text/html regardless of any file type that may be inferred by filename extensions! This is because the content type will refer to the server-generated file that contains the 404 error message. I do **not** want you to GET that file!

Q: What is a "redirect" and how do I handle it?

A: When you issue the HEAD command and get a response code in the range 300-399 (typically 301), this means the link is being redirected to another file. That file will be given in the "Location:" field in the response header. It will be a full URL, so needs to be fully processed (check for internal/external, then issue HEAD again).

Q: Pages can use cascading style sheets that themselves have images. Do I need to find these?

A: No. Just look for IMG and HREF (**not** case sensitive) in the current page.

Q: What if the same image file is referenced repeatedly in an HTML file?

A: Count each occurrence separately.

Q: What about HREF attributes containing "mailto" links?

A: They are included in count of "number of hyperlinks on a page", but otherwise not processed.

Q: GET requires that the pathname start with "/" and contain full path relative to server (necessary due to stateless server). But HTML assumes the browser will keep track of current directory and allows links to be relative to it. How do I deal with this?

A: Do what the browser does. Keep track of current directory, and piece together the file path using it plus the relative path. You need to handle "../" and "./" when they appear in a link.

Q: What if the hyperlink ends with a directory name or with "/"?

A: In the first case, server will respond with a redirect message(see earlier question). In the second case, server will fetch index.html, index.htm, default.htm, ... as appropriate.